| | Kompressionsalgorithmen | | AnPr |
|---|---|---|---|
| | Name | Klasse | Datum |

# 1 Situation

In der Computertechnik spielen Kompressionsalgorithmen eine große Rolle. Daten, welche in einer „nutzbaren“ Form, wie bspw. Bilddateien, vorliegen, bergen mitunter Redundanzen, welche durch einen geeigneten Algorithmus zugunsten eines geringeren Datenvolumens herausgenommen werden können. In dieser Übungsreihe möchten wir uns um solche Algorithmen kümmern. Hierbei sind zwei Dinge wichtig:

1. die von uns genutzten Algorithmen sind verlustfrei – das aus der Kompression wieder hergestellte Objekt ist vom originalen Objekt nicht zu unterscheiden.
2. Wir kümmern uns nur um einfache, rein algorithmisch umzusetzende Verfahren. Mathematische Verfahren (wie bspw. die „arithmetische Kodierung“) betrachten wir hier nicht.

Bevor wir in die Einzelthemen einsteigen, hier eine kurze Darstellung des Ziels. Wenn wir die nebenstehende Grafik als Bitmap Datei ansehen, können wir uns ein relativ einfaches Konzept für eine einfache Kompression vorstellen. Wir gehen Zeile für Zeile vor und merken uns einfach, wie oft die gleiche Farbe hintereinander vorkommt. Dadurch speichern wir weitaus weniger Informationen als bei einer Bitmap üblich – nämlich jede Pixelinformation einzeln. Dies ist (vor allem bei farbigen Dateien) immer noch sehr primitiv, aber die Grundprinzipien können wir hier relativ schön sehen.

*Achtung – diese Übungsreihe erstreckt sich von einfachen Algorithmen bis am Ende zu relativ komplexen Strukturen. Arbeiten Sie die Übungen Schritt für Schritt durch, arbeiten Sie sauber und nutzen Sie bei Fehlfunktionen den Debugger sinnvoll. Erst wenn Sie absolut nicht weiterkommen, können Sie einen Blick auf die Musterlösung unter [https://github.com/maikaicher/Algorithm](https://github.com/maikaicher/Algorithm) werfen.*

# 2 Transfer des Problems in Textdaten

Beginnen wir mit einem sehr viel einfacheren Problem – der Kompression von Buchstabentexten. Gehen wir davon aus, wir haben die Buchstaben a-z, A-Z, Leerzeichen und die Satzzeichen. Hierbei haben wir folgenden (zugegebenermaßen sinnlosen) String:

```
aaaabbxxx hhhaaaaaaallllllooo!
```

Wir könnten nun folgendes ansetzen: Wir zählen jedes Zeichen und schreiben anstatt `aaaa` nur `a`, gefolgt von der Anzahl – also `a4`. Der String müsste also nun lauten

```
a4b2x3 1h3a7l6o3!1
```

## 2.1 Aufgabe 1

Erstellen Sie einen Algorithmus, der den Text entsprechend den oben genannten Vorgaben komprimiert. Gehen Sie davon aus, dass bereits im Vorfeld sichergestellt wurde, dass nur die erlaubten Zeichen eingesetzt wurden. Nennen Sie die Klasse „`SimpleText`“ und die Methode:
„`public static String compress(String sIn)`“.
*Achten Sie bei der Umsetzung auf mögliche Sonder- oder Fehlerfälle! Geben Sie im Fehlerfall null zurück.*

## 2.2 Aufbabe 2

Ergänzen Sie nun die Klasse um eine Methode „`decompress(String sIn)`“, welche den String wieder zurückverwandelt. Prüfen Sie in der main-Methode, ob der zurückgewandelte String auch wirklich dem Originalstring entspricht.
*Achten Sie bei der Umsetzung auf mögliche Sonder- oder Fehlerfälle! Geben Sie im Fehlerfall null zurück.*

## 2.3 Auswertung

Sehen Sie sich zuerst das Errorhandling bei der Komprimierungs- und Dekomprimierungsfunktion an. Was fällt Ihnen auf:

Sehen Sie sich die Kompressionsrate $K = n_{decomp}/n_{comp}$ für den oben genannten String an. Wie hoch ist er:

Wie hoch wäre er bei folgendem String:

```
Hallo Welt
```

Was ist hier also das Problem?

# 3 Selektive Komprimierung

Wir können also davon ausgehen, dass erst ab drei aufeinanderfolgenden Zeichen die Kompression überhaupt sinnvoll ist. Insofern müssen wir unseren Algorithmus dergestalt anpassen, dass wir zuerst prüfen, wie oft ein Zeichen überhaupt vorkommt. Wenn die Anzahl größer drei ist, geben wir es als „Zeichen + Anzahl“ aus, ansonsten 1:1 die entsprechenden Zeichen wie im Original.

## 3.1 Aufgabe 3

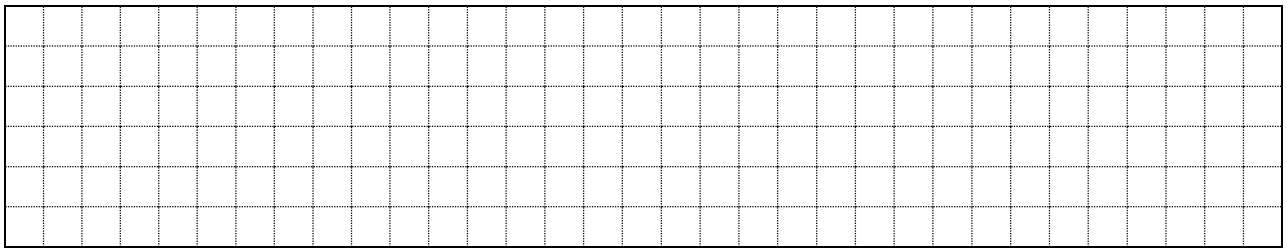
Erstellen Sie einen Algorithmus, der die neuen Vorgaben umsetzt. Gehen Sie auch hier davon aus, dass bereits im Vorfeld sichergestellt wurde, dass nur die erlaubten Zeichen eingesetzt wurden. Nennen Sie die Klasse „`SelectiveText`“ und die Methode:
„`public static String compress(String sIn)`“.
*Achten Sie bei der Umsetzung auf mögliche Sonder- oder Fehlerfälle! Geben Sie im Fehlerfall null zurück.*

## 3.2 Aufgabe 4

Wie Sie vermutlich festgestellt haben, konnte der ursprüngliche Komprimierungs-Algorithmus relativ schnell angepasst werden. Anders wird dies nun bei der Dekompression aussehen. Setzen Sie dies um:
*Achten Sie bei der Umsetzung auf mögliche Sonder- oder Fehlerfälle! Geben Sie im Fehlerfall null zurück.*

## 3.3 Auswertung

Warum benötigen wir jetzt kein Errorhandling mehr in der Dekompressionsmethode:

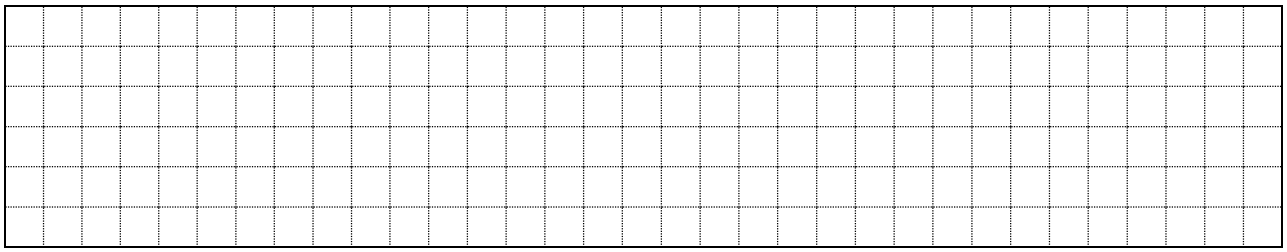


# 4 Erweiterung der erlaubten Zeichen

Versuchen Sie nun den folgenden String zu komprimieren und wieder zu dekomprimieren:

```
aaaabbxxx 4xhhhaaaaaaalllllllooo!
```

Warum erhalten Sie nun ein falsches Dekompressionsergebnis?



Eine mögliche Lösung für dieses Problem ist es, die Struktur des komprimierten Formats zu verändern. Beispielsweise könnte man die Zahlen, welche für die Anzahl der Zeichen stehen, zwischen zwei neu zu definierende Steuerzeichen setzen – beispielsweise Klammern:

```
a(4)bbx(3) 4xh(3)a(7)l(6)o(3)!
```

Mit dieser Struktur haben wir uns aber ein neues Problem geschaffen! Was ist, wenn wir in unserem ursprünglichen String Klammern haben – also bspw. die folgende Kombination:

```
aaaaaaab(4)
```

Unser Algorithmus würde dann vermutlich folgende Kompressionssequenz erstellen:

```
a(7)b(4)
```

Dieser würde dann aber zu folgendem Ergebnis führen:

```
aaaaaaabbbb
```

Wir müssen also die Klammern „escapen“. Wenn wir den Backslash als Escapezeichen verwenden, müsste der komprimierte String also wie folgt aussehen.

```
a(7)b\(4\)
```

Nun haben wir aber ein weiteres Zeichen als „Steuerzeichen“ definiert – den Backslash. Dieser muss nun auch im Kompressionsstring escaped werden. Dies gilt immer bei der Einführung eines Escapezeichens – wir müssen immer davon ausgehen, dass das Escapezeichen ebenfalls in den „Rohdaten“ vorkommen kann. Folgende Rohdatensequenz:

```
bbbbb\aaaaaaa\(4)
```

würde also zu folgendem Kompressionsstring werden:

```
b(5)\\a(7)\\\(4\)
```

Hieraus ergeben sich nun ein paar weitere Überlegungen. Zum einen ist die Anzahl von aufeinanderfolgenden gleichen Zeichen, die sinnvoll komprimiert werden können, auf 5 gewachsen. Bei 4 Zeichen ist der Originalstring und der komprimierte String gleich lang:

`aaaa` würde zu `a(4)` - also beide mit der Länge 4

Weiterhin haben wir drei Zeichen, welche escaped werden müssen: „(“ „)“ und „\“. Dies wiederum verschlechtert den Kompressionsgrad, wenn mindestens eines der drei Zeichen im Originalstring vorkommen. Besser wäre es also, wenn wir nur ein Steuerzeichen hätten. Wenn wir uns nun den oben als Beispiel aufgeführten String „`a(4)bbx(3) 4xh(3)a(7)l(6)o(3)!`“ nochmal vor Augen führen sehen wir, dass nach jeder öffnenden Klammer immer Ziffern, gefolgt von einer schließenden Klammer kommen. Insofern können wir auf die Unterscheidung von öffnender und schließender Klammer verzichten und fordern: „nach dem Steuerzeichen kommen Ziffern, gefolgt von einem Steuerzeichen“. Zur Abgrenzung nehmen wir nun das Pipe-Symbol als Steuerzeichen und setzen es für beide Klammern ein:

`a|4|bbx|3| 4xh|3|a|7|l|6|o|3|!`

Jede Zahl, welche also zwischen zwei Pipesymbolen steht, repräsentiert also die Anzahl des Zeichens, welches vor dem ersten der beiden Pipesymbolen steht. Da wir nun nur noch ein notwendiges Steuerzeichen haben, können wir es auch gleich zum escapen verwenden. Das doppelte Pipesymbol im komprimierten String steht also für ein einzelnes im Originalstring:

`bbbbb|aaaaaaa||4|`

würde also zu folgendem Kompressionsstring werden:

`b|5|||a|7|||||4||`

Dies gilt es nun, algorithmisch umzusetzen.

## 4.1 Aufgabe 5

Erstellen Sie nun den Algorithmus für die Kompression und die Dekompression für beliebige Strings unter Nutzung der oben genannten Überlegungen in der Klasse `EscapeText`. Machen Sie sich hierbei auch Gedanken zu möglichen Fehlerquellen im komprimierten String.

Wie hoch die Kompressionsrate für folgenden String: `aaaabbxxx hhhaaaaaaalllllooo!`

Warum ist das Verhältnis schlechter als bei Aufgabe 2.3?

# 5 Stringaufbau

Es gibt mehrere Möglichkeiten, einen String sequenziell aufzubauen. Die aus Performancesicht wohl schlechteste Lösung in Java ist es, ihn über Stringverkettung zu realisieren. Um dies zu prüfen, machen wir uns ein kleines Testprogramm, indem wir einen String mit 100.000 zufälligen Zeichen aufbauen:

```
s = ""
myRnd = new Random()
Zähle i von 0 bis < 100.000
    s += (char)myRnd.nextInt(256)
Ausgabe s
```

## 5.1 Aufgabe 6

Um die Performance von diesem Algorithmus zu testen, bauen Sie den oben genannten Algorithmus in eine statische Methode ein und geben den String s zurück (anstatt ihn in der Konsole auszugeben). Dann messen sie mit Hilfe von `System.currentTimeMillis()` die Ausführungszeit und notieren sie.

Wir erhalten eine durchaus signifikante Verarbeitungszeit für einen relativ einfachen Algorithmus. Dies liegt daran, dass das Stringobjekt „immutable" ist – also nicht veränderbar. Bei jedem Durchlauf von `s += …` wird ein neues Stringobjekt erzeugt und in gewissen Abständen die alten Stringobjekte über den Garbage Collector entsorgt.

## 5.2 Aufgabe 7

Um das Ganze nun zu beschleunigen, schreiben wir die Zufallswerte nicht in einen String, sondern in ein Character-Array „c" und wandeln es am Schluss mit `String.valueOf(c);` wieder in einen String um. Realisieren Sie dies und prüfen Sie auch hier die Ausführungszeit:

Das Problem für unseren Kompressionsalgorithmus ist jedoch, dass wir vorher die Größe des Character-Arrays wissen müssten – was jedoch nicht ohne weiteres möglich ist. Eine Alternative wäre dann die ArrayList, womit die Umwandlung in einen String nicht mehr über `String.valueOf(c);` realisieren könnten, sondern die ArrayList Daten zuerst in ein char-Array umkopieren und dann daraus den String generieren.

## 5.3 Aufgabe 8

Da die schrittweise Erzeugung eines Strings keine unübliche Aufgabe ist, finden wir in Java ein eigenes Objekt für diesen Taks - den `StringBuilder`. Dieser hat mit „`append`" die Funktion einzelne Elemente hinzuzufügen und kann mit „`.toString()`" einfach in einen String umgewandelt werden. Realisieren Sie nun den Algorithmus mit dem `StringBuilder` und messen wiederum die Ausführungszeit:

# 6 Reine Array-Lösung

Um unser ursprüngliches Ziel, Bitmaps zu komprimieren, zu realisieren, müssen wir uns die Array-Lösung für die Stringerzeugung nochmal genauer ansehen. Wir übernehmen also die einzelnen Character in ein Array. Da wir die Länge des komprimierten Ergebnisses nicht vorher wissen, müssen wir den Algorithmus mit einer `ArrayList` von Character realisieren und anschließend sie in ein `char`-Array umkopieren.

## 6.1 Aufgabe 9

Passen Sie den Algorithmus von Aufgabe 5 so an, dass wir keinen String erzeugen, sondern eine `ArrayList` von Character. Die Klasse `ArrayText` soll also zwei Methoden aufweisen:

```
char[] compress(char[] sIn)
char[] decompress(char[] sIn)
```

*Hinweis: Sie werden sich vermutlich leichter tun, wenn Sie die Zahl für die Character-Anzahl in der little-Endian Schreibweise hinterlegen. Also die Zahl „hundertachtundzwanzig" 128 in „verkehrter" Reihenfolge notieren: 821. Solange die Kompression und die Dekompression von der gleichen Notationsreihenfolge ausgehen, ist dies OK.*

# 7 Byte-Array

Wenn die Array-Lösung mit `char`-Arrays funktioniert, ist es ein relativ kleiner Schritt, dies in eine Lösung mit byte-Arrays umzuwandeln. Hierzu müssen lediglich alle `char` bzw. `Character` Datentypen in `byte` bzw.

`Byte` umgewandelt werden. Das Steuerzeichen, welches wir bis jetzt als Pipesymbol „|“ definiert haben, wird nun zu einem beliebigen Bytewert. Hier sollte man nicht unbedingt 0x00 oder 0xFF verwenden, da davon auszugehen ist, dass diese Werte in Files recht häufig vorkommen. Als Beispiel können Sie den Dezimalwert 100 (also hex 0x64) verwenden.

## 7.1 Aufgabe 10

Schreiben Sie eine Klasse BasicBinary, welche folgende Methoden beinhaltet:

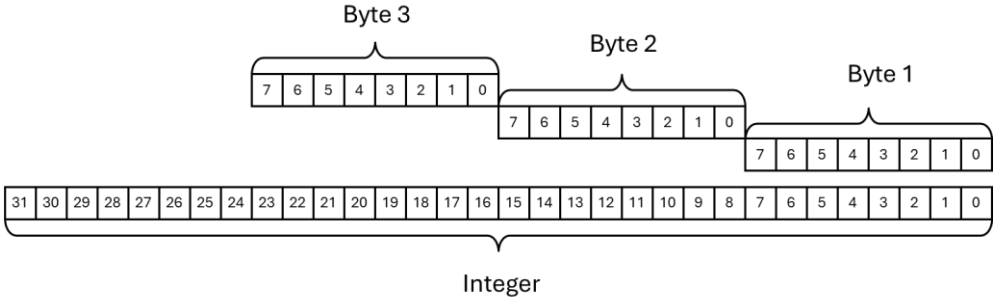
```
byte[] compress(byte[] sIn)
byte[] decompress(byte[] sIn)
void addNumberLittleEndian(ArrayList<Byte> al, int number)
byte[] readBinary(String fileName) throws IOException
boolean writeBinary(String fileName, byte[] data) throws IOException
```

Erstellen Sie ein Bitmap-File wie in Kapitel 1 gezeigt, lesen Sie es und schreiben die komprimierten Daten unter einem neuen Namen zurück auf das Filesystem. Lesen Sie danach dieses File wieder, dekomprimieren Sie es und schreiben Sie es wieder unter einem neuen Namen (als *.bmp Datei) auf das Filesystem und öffnen es mit „Paint“ um zu prüfen, ob die Komprimierung funktioniert hat. Bestimmen Sie den Komprimierungsgrad:

Ändern Sie nun das Bild, indem Sie den Smiley und den Hintergrund in einem orange-Ton einfärben. Ermitteln Sie nun wieder den Komprimierungsgrad:

Warum ist das Verhältnis nun um so viel schlechter?

Um die Kompressionsrate bei Farben ungleich Weiß oder Schwarz (bzw. eigentlich ungleich allen Grautönen) zu erhöhen, müssen wir also drei Bytes auf einmal heranziehen und vergleichen. Eine Möglichkeit, dies zu tun ist, die Daten nicht in ein byte-Array zu laden, sondern in ein int-Array, bei dem wir „nur“ 24 Bits pro Integerwert verwenden, indem wir drei Bytes auf die rechten 24 Bits legen.




Hierbei müssen wir aber nun darauf achten, dass wir die Anzahl der gesamten Bytes wissen. Wenn wir bspw. 1000 Bytes im File haben, werden sie in 3er Pakete zusammengefasst. Dadurch sind in 333 Integer-Arraypositionen die drei Bytes voll belegt und im letzten, dem 334ten Eintrag nur eines (333x3 = 999; 999 + 1 = 1000). Wir müssen also diese Information transportieren. Eine recht simple Lösung ist es, im 4. Byte des Integer-

Eintrages zu hinterlegen, wie viele Bytes jeweils zu finden sind. Hierfür kann man die Bits 24 und 25 verwenden.

## 7.2 Aufgabe 11

Übernehmen Sie den Code der Klasse `ArrayText` in eine neue Klasse `TrippleArrayText` und ändern Sie ihn so um, dass sie jeweils Dreiergruppen von Character als zusammengehörige Daten interpretieren. Hierbei gehen wir davon aus, dass wir ausschließlich Character verwenden, welche nur ein Byte belegen. Folgender Beispiel-String:

abcabcabcabcabca||xyzxyzxyzxyz|23|23|23|23

sollte dann wie folgt komprimiert werden:

abc|5|a||||xyz|4|||23|4|

Dieser String ist wie folgt zu interpetieren:
abc kommt 5x hintereinander vor, dann ein a gefolgt von zwei | Symbolen (die ja escaped werden). Danach xyz 4x, gefolgt von |23 jeweils 4x.

*Hinweis: Es ist sinnvoll, Teilaspekte des Algorithmus in ein eigenes Unterprogramm zu verlagern. In meinem Code war es bspw. hilfreich, eine Methode zu erstellen, welche eine int-Zahl (mit den drei char-Werten) in der little-Endian Reihenfolge in ein Dynamisches Array einreiht. Eine weitere hilfreiche Funktion war eine Extraktionsmethode, welche aus einer Integerzahl die Character extrahiert.*

## 7.3 Aufgabe 12

Erstellen Sie nun auf Basis von `TrippleArrayText` eine Klasse `RgbBinary`, welche das gleiche Verfahren anwendet, jedoch nur für Bytes.

Ermitteln Sie nun wieder den Komprimierungsgrad der Datei mit dem orangenen Hintergrund:

## 7.4 Aufgabe 13

Eine letzte mögliche Optimierungsoption haben wir noch. Das Steuerbyte, welches wir zum escapen verwenden wurde „beliebig“ gesetzt. Da aber jedes Byte in der Originaldatei, welches dem escape-Byte entspricht, zwei Bytes an Platz benötigt und somit die Effizienz der Kompression reduziert, können wir vor dem Codieren prüfen, welcher Wert in der Rohdatei am seltensten vorkommt und dieses dann dynamisch als Steuerzeichen setzen. Dadurch müssen wir aber das genutzte Byte als Wert in der Ausgabedatei hinterlegen, damit wir beim Dekomprimieren den richtigen Wert verwenden. Am sinnvollsten ist es, in der Kompressionsdatei das genutzte Steuerzeichen gleich als erstes Byte zu hinterlegen.

Realisieren Sie nun diesen Algorithmus und prüfen Sie den Komprimierungsgrad:

Warum ist er (vermutlich bei Ihnen) etwas schlechter als ohne die „Optimierung“?

Ändern Sie nun die Hintergrundfarbe des Bildes auf den Grauwert 0x646464 und vergleichen Sie den Kompressionsgrad mit und ohne die Optimierung.